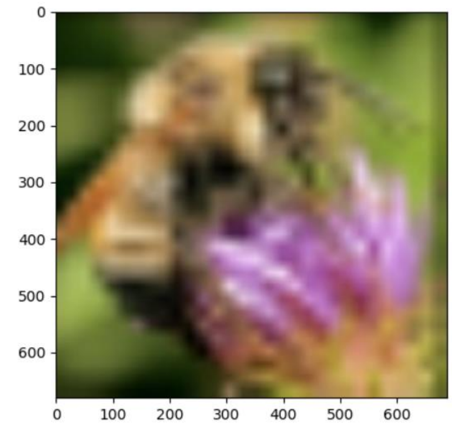
```
import numpy as np
import cv2
import math
from matplotlib import pyplot as plt
import cv2.xfeatures2d
import scipy
```

1. Up-sampling
    a.

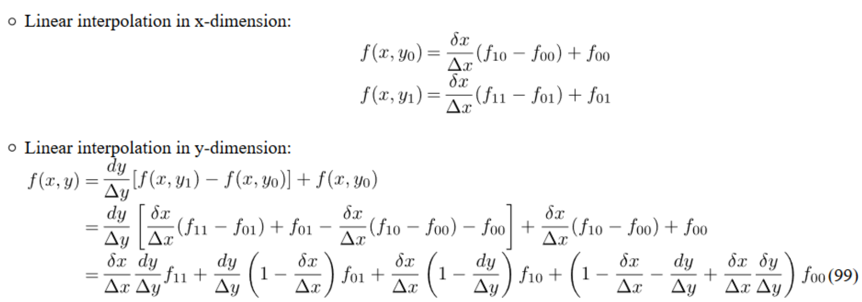
```
def linear_interpolation(img, size):
    shape = img.shape
    result = np.ones((int(shape[0]), int(shape[1] * size), shape[2]))

    for i in range(shape[0]):
        for j in range(shape[1] * size):
            if j % size == 0:
                result[i][j] = img[int(i)][int(j / size)]
            else:
                if math.ceil(i) < shape[0] and math.ceil(j / size) <
shape[1]:
                    result[i][j] = ((math.ceil(j / size) - j / size) /
(math.ceil(j / size) - math.floor(j / size))) * \
                                   img[i][math.floor(j / size)] + ((j /
size - math.floor(j / size)) / (
                            math.ceil(j / size) - math.floor(j / size)))
* img[i][math.ceil(j / size)]
                else:
                    result[i][j] = img[i][math.floor(j / size)]
    shape = result.shape
    result1 = np.ones((int(shape[0] * size), int(shape[1]), shape[2]))
    for i in range(shape[0] * size):
        for j in range(shape[1]):
            if i % size == 0:
                result1[i][j] = result[int(i / size)][j]
            else:
                if math.ceil(i / size) < shape[0] and math.ceil(j) <
shape[1]:

                    result1[i][j] = ((math.ceil(i / size) - i / size) /
(math.ceil(i / size) - math.floor(i / size))) * \
                                    result[math.floor(i / size)][j] +
((i / size - math.floor(i / size)) / (
                            math.ceil(i / size) - math.floor(i / size)))
* result[math.ceil(i / size)][j]
                else:
                    result1[i][j] = result[math.floor(i / size)][j]

    return result1
```

```
image = cv2.imread("bee.jpg")
image = image.astype(np.float32) / 255
image1 = cv2.cvtColor(image,
cv2.COLOR_BGR2RGB)
result = linear_interpolation(image1, 4)
result = np.round(result *
255).astype(np.uint8)
plt.imshow(result)
plt.show()
```



This is a 1D filter applied twice. I think we can do the operation with a 2D filter. The filter is a Bilinear Interpolation filter.

b.



- Linear interpolation in x-dimension:

$$f(x, y_0) = \frac{\delta x}{\Delta x}(f_{10} - f_{00}) + f_{00}$$
$$f(x, y_1) = \frac{\delta x}{\Delta x}(f_{11} - f_{01}) + f_{01}$$

- Linear interpolation in y-dimension:

$$\begin{aligned} f(x, y) &= \frac{dy}{\Delta y}[f(x, y_1) - f(x, y_0)] + f(x, y_0) \\ &= \frac{dy}{\Delta y}\left[\frac{\delta x}{\Delta x}(f_{11} - f_{01}) + f_{01} - \frac{\delta x}{\Delta x}(f_{10} - f_{00}) - f_{00}\right] + \frac{\delta x}{\Delta x}(f_{10} - f_{00}) + f_{00} \\ &= \frac{\delta x}{\Delta x}\frac{dy}{\Delta y}f_{11} + \frac{dy}{\Delta y}\left(1 - \frac{\delta x}{\Delta x}\right)f_{01} + \frac{\delta x}{\Delta x}\left(1 - \frac{dy}{\Delta y}\right)f_{10} + \left(1 - \frac{\delta x}{\Delta x} - \frac{dy}{\Delta y} + \frac{\delta x}{\Delta x}\frac{\delta y}{\Delta y}\right)f_{00} \quad (99) \end{aligned}$$

I found this algorithm for bilinear interpolation, but I could not implement it.

2. Interest point detection

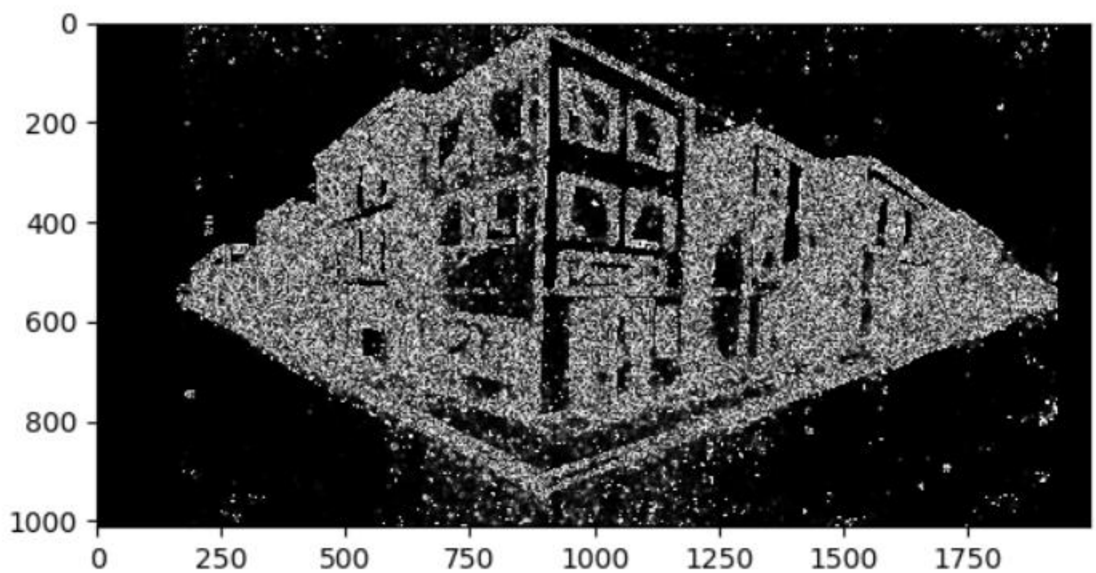
a.

```
def corner(img, a, key):
    blur = cv2.GaussianBlur(img, (5, 5), 7)
    Ix = cv2.Sobel(blur, cv2.CV_64F, 1, 0, ksize=5)
    Iy = cv2.Sobel(blur, cv2.CV_64F, 0, 1, ksize=5)
    IxIy = np.multiply(Ix, Iy)
    Ix2 = np.multiply(Ix, Ix)
    Iy2 = np.multiply(Iy, Iy)
    Ix2_blur = cv2.GaussianBlur(Ix2, (7, 7), 10)
    Iy2_blur = cv2.GaussianBlur(Iy2, (7, 7), 10)
    IxIy_blur = cv2.GaussianBlur(IxIy, (7, 7), 10)
    det = np.multiply(Ix2_blur, Iy2_blur) - np.multiply(IxIy_blur,
IxIy_blur)
    trace = Ix2_blur + Iy2_blur
    if key == "harris":
        R = det - a * np.multiply(trace, trace)
    elif key == "brown":
        with np.errstate(divide='ignore', invalid='ignore'):
            R = det / trace
    return R
```

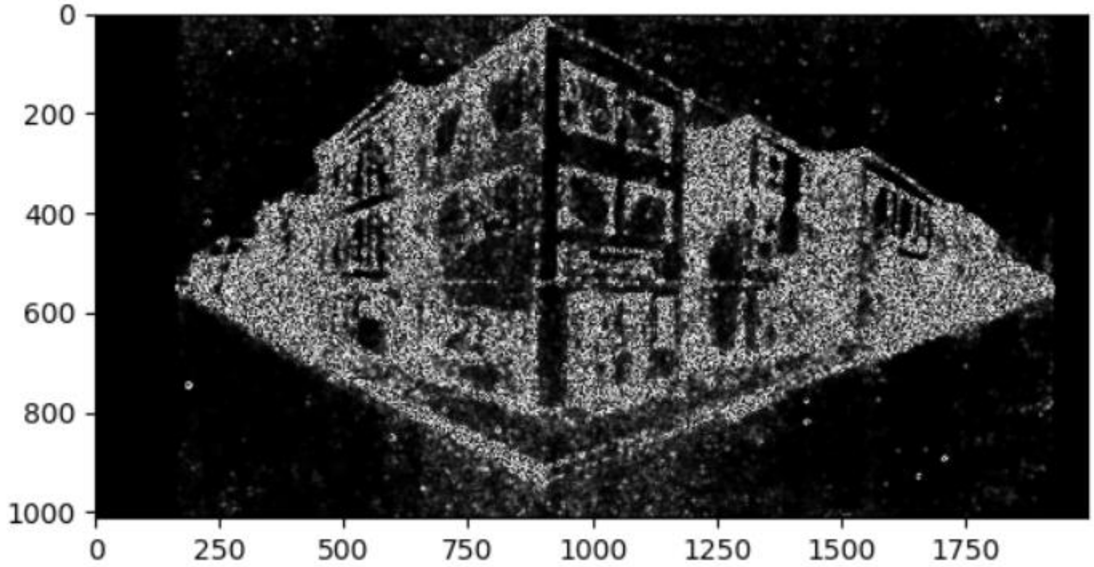
```
image = cv2.imread("building.jpg")
image = image.astype(np.float32) / 255
image1 = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
result = corner(image1, 0.05, "harris")
result1 = corner(image1, 0.05, "brown")
result = np.round(result * 255).astype(np.uint8)
result1 = np.round(result1 * 255).astype(np.uint8)
plt.imshow(result)
plt.show()
plt.imshow(result1)
plt.show()
```

Output:

harris:



brown:

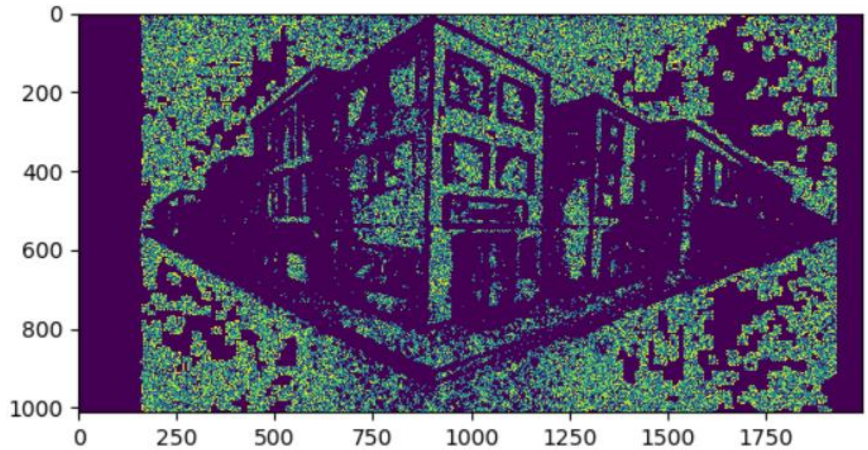


For harris detector, you can tone the strength of the detector by changing alpha, for brown you cannot.

I think there is a way to detect corners without using determinant and trace, if we use eigenvalues instead.

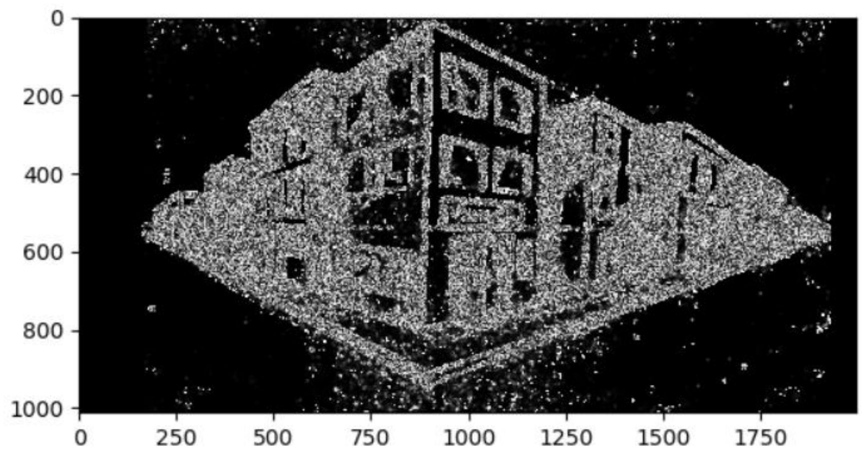
```
def corner_eigen(img):
    eig = cv2.cornerEigenValsAndVecs(img, 7, 7)
    result = np.empty(img.shape, dtype=np.float32)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            lambda_1 = eig[i, j, 0]
            lambda_2 = eig[i, j, 1]
            result[i, j] = lambda_1 * lambda_2 - 0.05 * pow((lambda_1 + lambda_2), 2)
    return result

image1 = cv2.imread("building.jpg", cv2.IMREAD_GRAYSCALE).astype(np.float32)
result = corner_eigen(image1)
result = np.round(result * 255).astype(np.uint8)
plt.imshow(result)
plt.show()
```

Output:

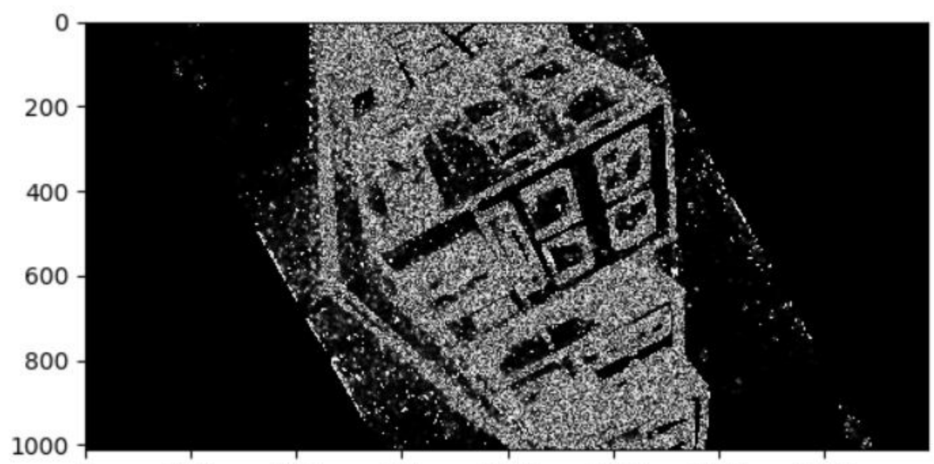


b.

Normal:



Rotated:



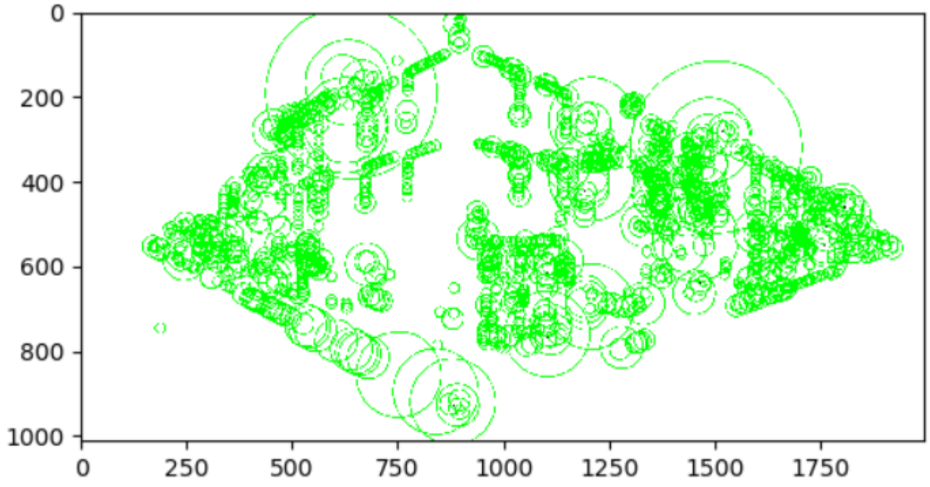
The corners all rotated by the same degree because harris corner detector is rotation invariant.

c.

```
def patch_view(img, patch_h, patch_w):
    h, w = np.array(img.shape) - np.array([patch_h, patch_w]) + 1
    return np.lib.stride_tricks.as_strided(np.ascontiguousarray(img), shape=(h, w, 
patch_h, patch_w),
                                           strides=img.strides + img.strides, 
writeable=False)


def suppression(img, neighbours):
    img_pad = np.pad(img, ((1, 1), (1, 1)), mode='constant', constant_values=0)
    neighbours_pad = []
    for item in neighbours:
        neighbours_pad.append(np.pad(item, ((1, 1), (1, 1)), mode='constant', 
constant_values=0))

    # img_patches = skl.extract_patches_2d(img_pad, (3, 3), min(h,w))

    img_patches = patch_view(img_pad, 3, 3)
    neighbours_patches = []
    for item in neighbours_pad:
        neighbours_patches.append(patch_view(item, 3, 3))

    patch_max = np.amax(img_patches, axis=(2, 3))

    neighbours_patches_maxes = []
    for item in neighbours_patches:
        neighbours_patches_maxes.append(np.amax(item, axis=(2, 3)))
    all_max = np.amax(np.dstack([patch_max] + neighbours_patches_maxes), axis=2)

    is_cur_max = np.equal(img, all_max)

    return img * is_cur_max


def find_max(img, length, sigma):
    scale = 2 ** (1 / length)
    lapace = []
    blur = []
    for i in range(length):
        blur_width = sigma * (scale ** i)
        lapace.append(cv2.GaussianBlur(img, (5, 5), sigma))
        blur.append(blur_width)

    counter = 0
    for item in lapace:
        lap = np.abs(cv2.Laplacian(item, ddepth=cv2.CV_32F, ksize=5, scale=1))
        lapace[counter] = lap
        counter += 1
    suppressed = [suppression(lapace[0], [lapace[1]])]
    for i in range(len(lapace) - 2):
```

```python
        (down, cur, up) = lapace[i:i + 3]
        suppressed += ([suppression(cur, [down, up])])
    suppressed += ([suppression(lapace[-1], [lapace[-2]])])

    stacked = np.dstack(suppressed)
    max_id = np.argmax(stacked, axis=2)
    max_item = np.amax(stacked, axis=2)
    blur = np.array(blur)
    corr_blur = blur[max_id]

    max_coords = np.nonzero(max_item)
    max_blur = corr_blur[max_coords]
    responses = max_item[max_coords]
    return np.array([responses, *max_coords, max_blur]).transpose()


def SIFT(img, length, sigma, threshold):
    result = np.dstack([np.copy(img)] * 3)
    shortest_side = min(img.shape[0], img.shape[1])
    reductions = int(np.log2(shortest_side)) - 1
    cur_image = img
    pyramids = []

    for i in range(reductions):
        scale = 2 ** i
        pyramids.append([cur_image, scale])
        cur_image = cv2.pyrDown(cur_image)

    max = []

    for img, scale in pyramids:
        oct_max = find_max(img, length, sigma)
        oct_max[:, 1:] *= scale
        max += [oct_max]
    max = np.concatenate(max)
    threshold1 = np.percentile(max[:, 0], threshold)
    for (response, y, x, blur_width) in max:
        if response > threshold1:
            radius = blur_width * (2 ** 0.5)
            cv2.circle(result, (int(x), int(y)), int(radius), (0, 255, 0), 2)

    return result


image1 = cv2.imread("building.jpg", cv2.IMREAD_GRAYSCALE).astype(np.float32)
result = SIFT(image1, 5, 9, 98)
plt.imshow(result)
plt.show()
```

Output:



d.
speeded up robust features (SURF)

Inspired by SIFT, but is faster. SURF uses an integer approximation of the determinant of Hessian blob detector, its feature descriptor is based on the sum of the Haar wavelet response around the point of interest.

Steps:

i. Feature Extraction
   1. Uses a basic Hessian matrix approximation on the image
   2. Filter with gaussian kernel to adapt to different scale
   3. Get the determinant of the Hessian matrix by using box filter to approximate gaussian second derivative.
   4. Analyze the scale space by up-scaling the filter size
   5. Non-maximum suppression

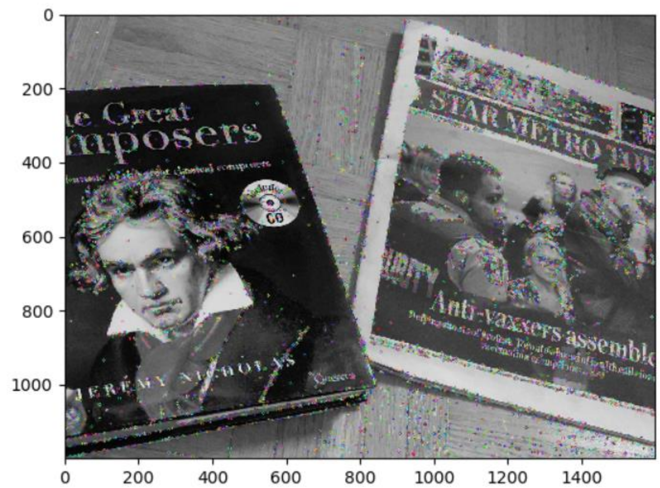
ii. Feature Description
   1. Calculate the Haar-wavelet responses in x and y-direction in a radius of 6s
   2. Using the vertical and horizontal wavelet responses to get the orientation with the largest sum value, this is the main orientation
   3. Extract the descriptor

3. Laplacian of Gaussian
   a.
   b.
4. SIFT Matching (using OpenCV SIFT implimentation)
   a.

```
def SIFT_opencv(img):
    sift = cv2.xfeatures2d.SIFT_create()
    kp, des = sift.detectAndCompute(img, None)
    result = cv2.drawKeypoints(img, kp, None)
    return [result, kp, des]


sample1 = cv2.imread("sample1.jpg", cv2.IMREAD_GRAYSCALE)
sample2 = cv2.imread("sample2.jpg", cv2.IMREAD_GRAYSCALE)
```

```
result = SIFT_opencv(sample1)
result1 = SIFT_opencv(sample2)
plt.imshow(result[0])
plt.show()
plt.imshow(result1[0])
plt.show()
```

Output:

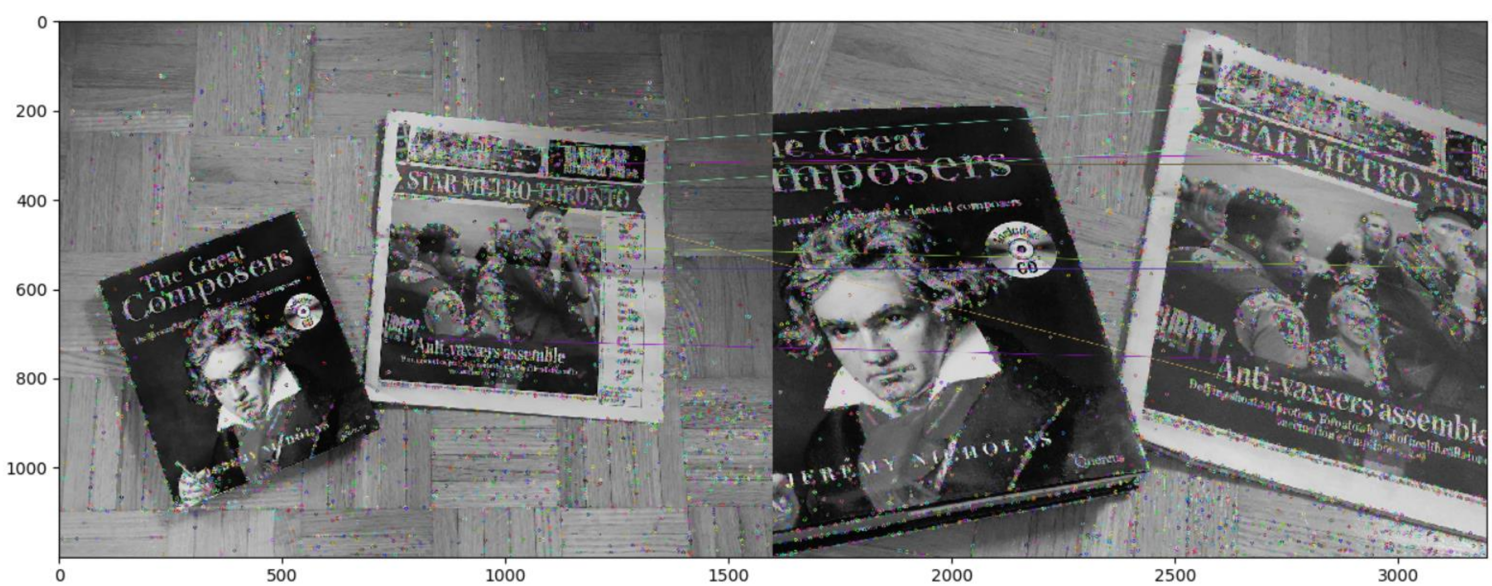


b.

```
def SIFT_matching(img1, img2, threshold):
    if img1[2] is None or len(img1[2]) == 0 or img2[2] is None or len(img2[2])
== 0:
        return cv2.drawMatches(img1[0], img1[1], img2[0], img2[1], [],
img2[0], flags=2)
# it is possible that the following step might run out of space, if so, need
more memory available
    euclidean = scipy.spatial.distance.cdist(img1[2], img2[2],
metric='euclidean')

    sorted1 = np.argsort(euclidean, axis=1)
    closest, closest1 = sorted1[:, 0], sorted1[:, 1]
    left_id = np.arange(img1[2].shape[0])
    dist_ratios = euclidean[left_id, closest] / euclidean[left_id, closest1]
    suppressed = dist_ratios * (dist_ratios < threshold)
    left_id = np.nonzero(suppressed)[0]
    right_id = closest[left_id]
    pairs = np.stack((left_id, right_id)).transpose()
    pair_dists = euclidean[pairs[:, 0], pairs[:, 1]]
    sorted_dist_id = np.argsort(pair_dists)
    sorted_pairs = pairs[sorted_dist_id]
    sorted_dists = pair_dists[sorted_dist_id].reshape((sorted_pairs.shape[0],
1))

    matches = []
    print("Best 10 values: ")
    for i in range(10):
        print("Pairs: {}, dist: {}".format(sorted_pairs[-i], sorted_dists[-
```

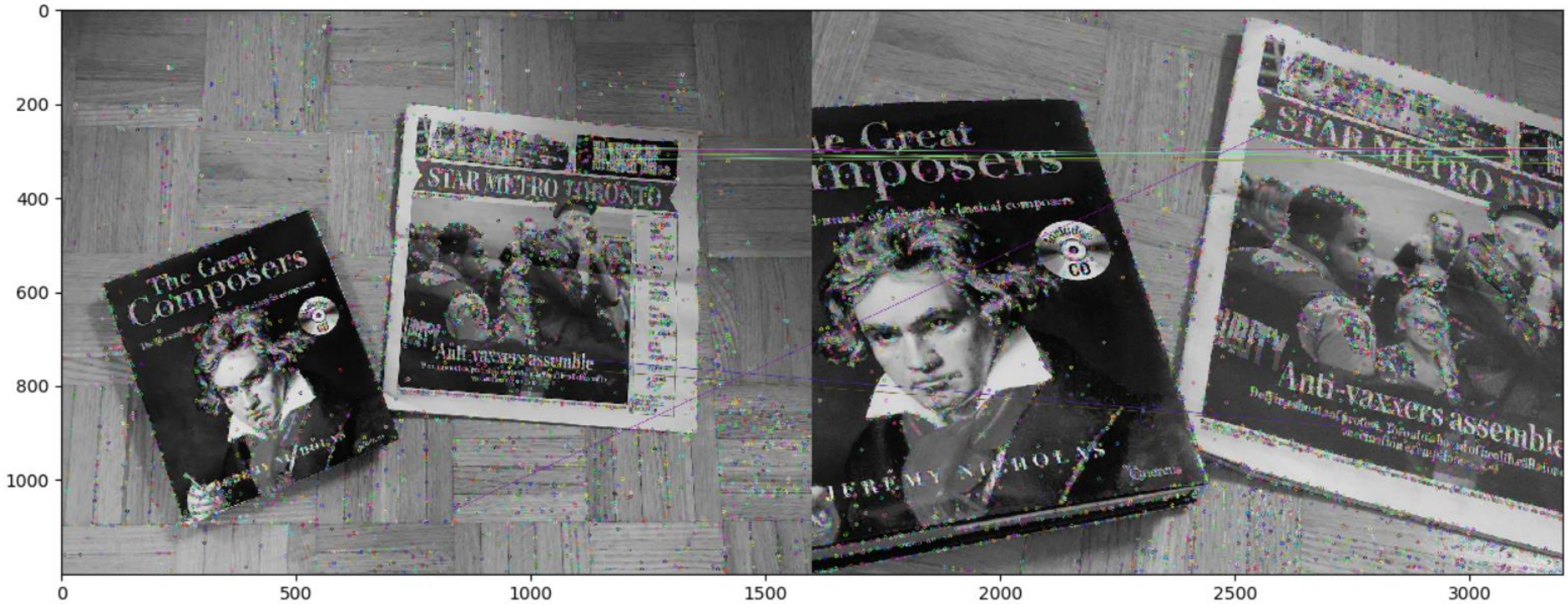
```
i]))
            matches.append(cv2.DMatch(sorted_pairs[-i][0], sorted_pairs[-i][1],
sorted_dists[-i]))
    result = cv2.drawMatches(img1[0], img1[1], img2[0], img2[1], matches,
img2[0], flags=2)
    return result

sample1 = cv2.imread("sample1.jpg", cv2.IMREAD_GRAYSCALE)
sample2 = cv2.imread("sample2.jpg", cv2.IMREAD_GRAYSCALE)
result = SIFT_opencv(sample1)
result1 = SIFT_opencv(sample2)
result2 = SIFT_matching(result, result1, 0.8)
plt.imshow(result[0])
plt.show()
plt.imshow(result1[0])
plt.show()
plt.imshow(result2)
plt.show()
```

Output:
Best 10 values:
Pairs: [5204 2234], dist: [29.71531592]
Pairs: [5249 8925], dist: [289.92930173]
Pairs: [5462 8906], dist: [286.95470026]
Pairs: [3255 5301], dist: [286.69670385]
Pairs: [3213 5800], dist: [280.25167261]
Pairs: [3348 5481], dist: [279.4548264]
Pairs: [2900 4441], dist: [276.50678111]
Pairs: [3624 5979], dist: [275.21264506]
Pairs: [3109 4952], dist: [274.58878346]
Pairs: [5112 8811], dist: [273.9069185]



c. L1-norm:

Change `metric='euclidean'` to `metric='cityblock'`



Best 10 values:
Pairs: [5204 2234], dist: [223.]
Pairs: [4029 6837], dist: [2348.]
Pairs: [5617 8966], dist: [2338.]
Pairs: [5249 8925], dist: [2281.]
Pairs: [5247 8906], dist: [2265.]
Pairs: [5462 8906], dist: [2261.]
Pairs: [5684 8985], dist: [2248.]
Pairs: [4786 6837], dist: [2126.]
Pairs: [3202 5429], dist: [2092.]
Pairs: [6113 9028], dist: [2075.]

L3-norm:
Change `metric='euclidean'` to `metric='minkowski', p=3`

Best 10 values:
Pairs: [5204 2234], dist: [16.96764318]
Pairs: [705 1290], dist: [173.51916173]
Pairs: [4595 7910], dist: [163.20813489]
Pairs: [4228 7132], dist: [160.42394083]
Pairs: [6113 9028], dist: [160.07841209]
Pairs: [5220 9052], dist: [159.68275681]
Pairs: [2984 4686], dist: [158.69855733]
Pairs: [3033 4876], dist: [158.63301568]
Pairs: [641 5808], dist: [157.38816954]
Pairs: [5462 8906], dist: [156.9600423]

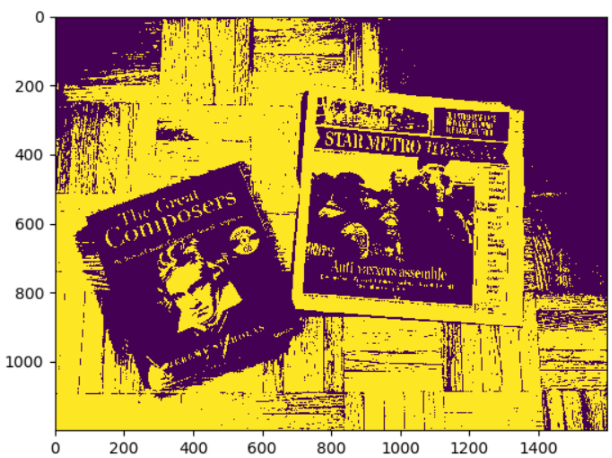
d.

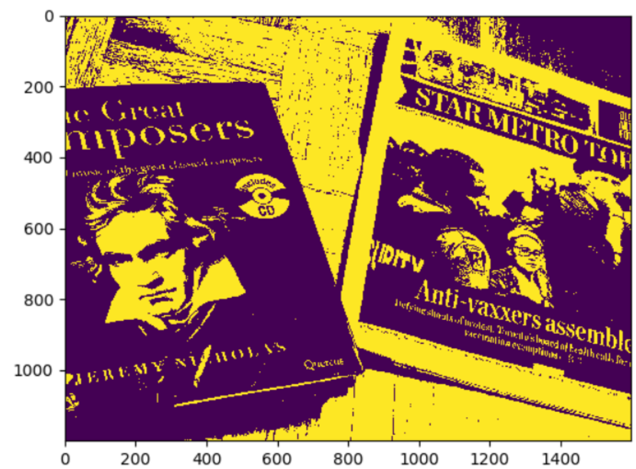
```
def add_noise(img):
    noise = np.random.normal(0, 0.08, img.shape[:2])
    result = img + noise
    result = result.astype(np.uint8)
    return result
sample1 = cv2.imread("sample1.jpg", cv2.IMREAD_GRAYSCALE)
sample2 = cv2.imread("sample2.jpg", cv2.IMREAD_GRAYSCALE)
sample1 = cv2.normalize(sample1, sample1, 0, 1, cv2.NORM_MINMAX)
sample2 = cv2.normalize(sample2, sample2, 0, 1, cv2.NORM_MINMAX)
plt.imshow(sample1)
plt.show()
plt.imshow(sample2)
plt.show()
sample1 = add_noise(sample1)
sample2 = add_noise(sample2)
plt.imshow(sample1)
plt.show()
```

```
plt.imshow(sample2)
plt.show()
# opencv's SIFT algorithm only produces entirely black image if the following step is
# not performed
sample1 = np.round(sample1 * 255).astype(np.uint8)
sample2 = np.round(sample2 * 255).astype(np.uint8)
result = SIFT_opencv(sample1)
result1 = SIFT_opencv(sample2)
result2 = SIFT_matching(result, result1, 0.8)
plt.imshow(result2)
plt.show()
```

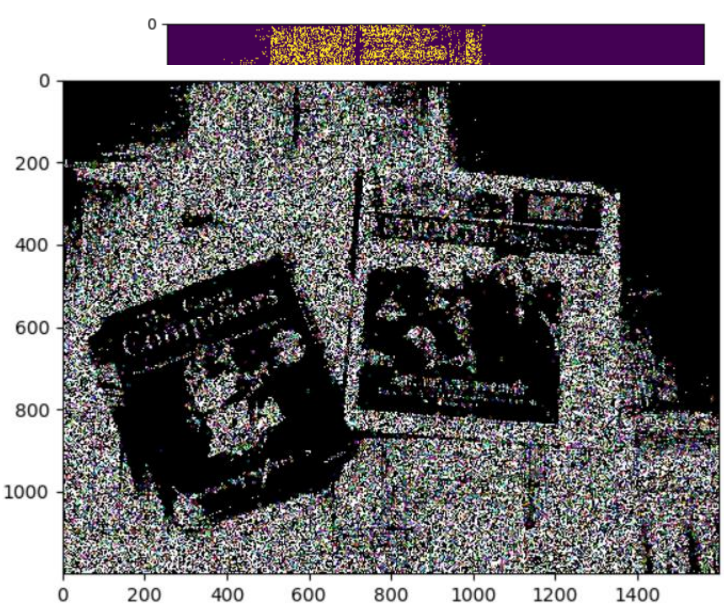
output:

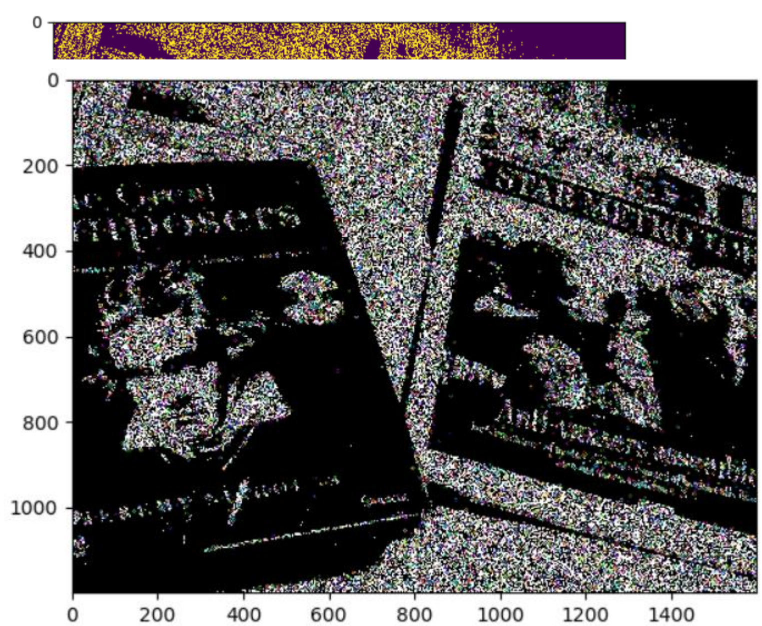
normalized:





Added noise:





After SIFT:

I could not run SIFT matching because it requires too much memory.
MemoryError: Unable to allocate array with shape (16728, 12463) and data type float64
The feature point detection detected too many noises as feature points. I hypothesize that if I were able to run the matching algorithm, the features would be matched very poorly due to the noises.

e.

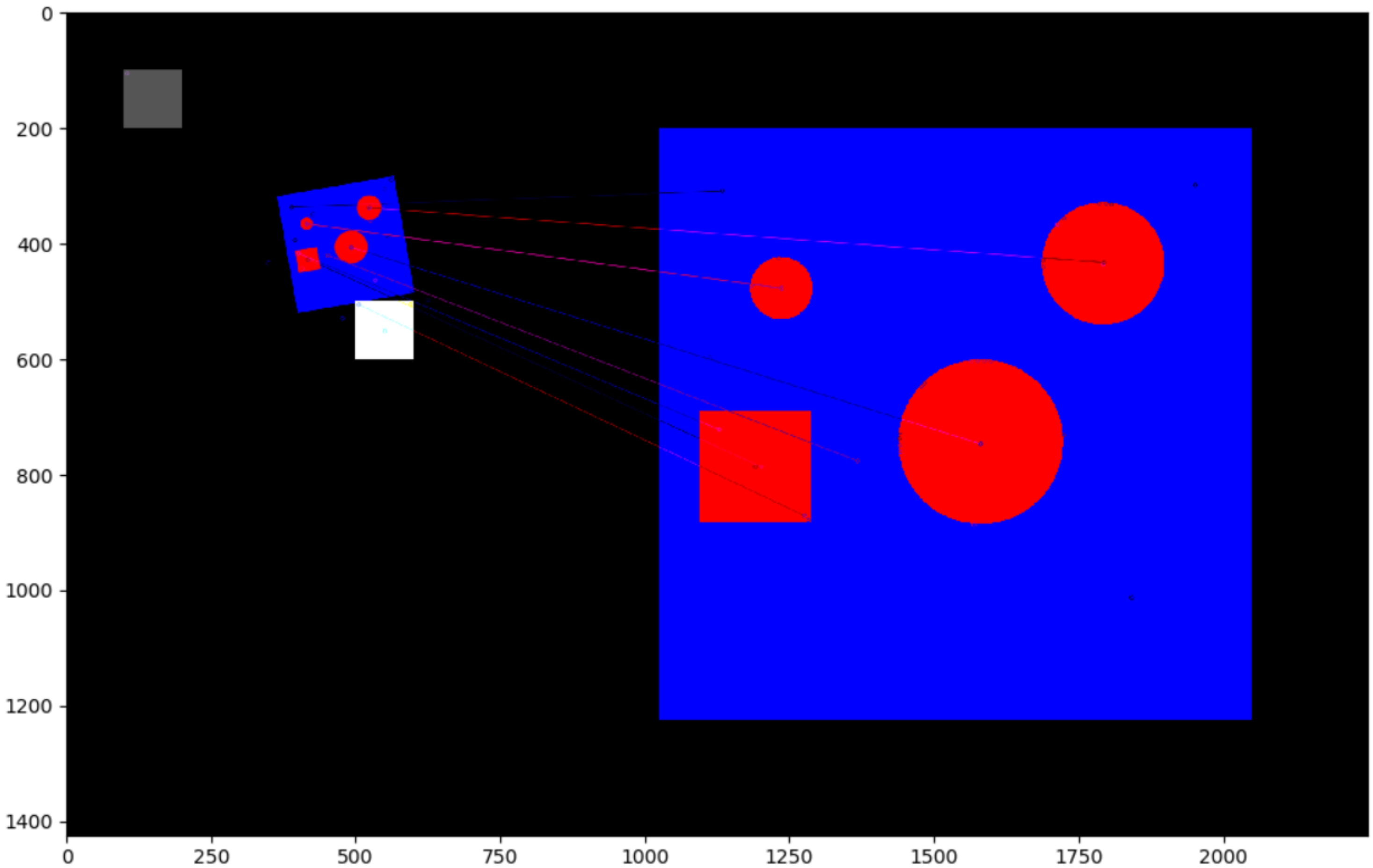
```
col1 = cv2.imread("colourSearch.png")
col2 = cv2.imread("colourTemplate.png")
b, g, r = cv2.split(col1)
b1, g1, r1 = cv2.split(col2)
result = [SIFT_opencv(item) for item in [b, g, r]]
result1 = [SIFT_opencv(item) for item in [b1, g1, r1]]

result2 = [SIFT_matching(result[i], result1[i], 0.8) for i in range(3)]
for i in range(3):
    result2[i] = cv2.cvtColor(result2[i], cv2.COLOR_BGR2GRAY)
result3 = cv2.merge(result2)

plt.imshow(result3)
plt.show()
```

Output:

Best 10 values:
Pairs: [19 13], dist: [21.09502311]
Pairs: [13 17], dist: [187.57931656]
Pairs: [3 6], dist: [184.49661244]
Pairs: [8 11], dist: [177.35557505]
Pairs: [7 10], dist: [172.58331321]
Pairs: [5 8], dist: [93.978721]
Pairs: [21 40], dist: [93.47192092]
Pairs: [25 38], dist: [75.41883054]
Pairs: [12 16], dist: [63.67102952]
Pairs: [20 39], dist: [62.8251542]

a different channel
Best 10 values:
Pairs: [4 2], dist: [23.60084744]
Pairs: [20 5], dist: [318.55768708]
Pairs: [11 9], dist: [247.24077334]
Pairs: [12 10], dist: [211.26996947]
Pairs: [15 11], dist: [193.86851214]
Pairs: [23 19], dist: [193.82208337]
Pairs: [7 1], dist: [191.9479096]
Pairs: [29 31], dist: [183.50476833]
Pairs: [16 12], dist: [164.50227962]

Pairs: [30 29], dist: [159.68719423]



I separated the three channels and ran the SIFT + matching algorithm separately for each channel. Then I merged the 3 channels back together. One of the channels was empty, so it did not have any matches.